hacs\textsc{spec}

a gateway to high-assurance cryptography

Franziskus Kiefer, Karthikeyan Bhargavan

Bas Spitters, Lasse Letager Hansen
Manuel Barbosa, Pierre-Yves Strub
Lucas Franceschino, Denis Merigoux
CRYPTO LIBRARIES

OpenSSL
NSS
BoringSSL
Lang
Web
OS
IoT
wolfSSL
Mbed TLS

Cryptography and SSL/TLS Toolkit
VERIFIED CRYPTO LIBRARIES

- **HACL***
  - Portable C
  - EC, AEAD, DH, Hash, Sig, PQC
- **F***
- **Coq**
- **AUCurves**
  - Rust
  - EC, BLS
- **Fiat**
  - C, Rust, Go
  - Field Arith
- **Cryptol/SAW**
  - C, Java
  - EC, AEAD, Hash, PQC
- **SAT/SMT**
Good news: For any modern crypto algorithm, there is probably a verified implementation.

But... research code with low-level APIs, and specs written in unfamiliar formal languages.
ChaCha20 and Poly1305 for IETF Protocols

Abstract

This document defines the ChaCha20 stream cipher of the Poly1305 authenticator, both as standard, "combined mode", or Authenticated Encryption.
let line (a:idx) (b:idx) (d:idx) (s:rotval U32) (m:state) : Tot state =
let m = m.[a] - (m.[a] +. m.[b]) in
let m = m.[d] - ((m.[d] \^ \ m.[a]) <<< s) in m

let quarter_round a b c d : Tot shuffle =
line a b d (size 16) @
line c d b (size 12) @
line a b d (size 8) @
line c d b (size 7)

proc chacha20_line(a : int, b : int, d : int, s : int, st : State) = {
var state;
state <- st;
state.[a] <- ((state).[a]) + ((state).[b]);
state.[d] <- ((state).[d]) `^` ((state).[a]);
state.[d] <- rotate_left ((state).[d]) (s);
return state;
}

proc chacha20_quarter_round(a : int, b : int, c : int, d : int, st : State) = {
var state;
state @@ chacha20_line (a, b, d, 16, st);
state @@ chacha20_line (c, d, b, 12, state);
state @@ chacha20_line (a, b, d, 8, state);
state @@ chacha20_line (c, d, b, 7, state);
return state;
}
F* Implementation

```fsharp
let line st a b d r =
    let sta = st.(a) in
    let stb = st.(b) in
    let std = st.(d) in
    let sta = sta +. stb in
    let std = std +. sta in
    let std = rotate_left std r in
    st.(a) <- sta;
    st.(d) <- std

let quarter_round st a b c d =
    line st a b d (size 16);
    line st c d b (size 12);
    line st a b d (size 8);
    line st c d b (size 7)
```

Translate

Portable C Code

```c
static inline void quarter_round(uint32_t *st, uint32_t a, uint32_t b, uint32_t c, uint32_t d) {
    uint32_t sta = st[a];
    uint32_t stb0 = st[b];
    uint32_t std0 = st[d];
    uint32_t sta10 = sta + stb0;
    uint32_t std10 = std0 + sta10;
    uint32_t std2 = std10 << (uint32_t)16 | std10 >> (uint32_t)16;
    st[a] = sta10;
    st[d] = std2;
    ...
```
inline fn __line_ref(reg u32[16] k, inline int a b c r) -> reg u32[16]
{
    k[a] ^= k[b];
    k[c] ^= k[a];
    _, _, k[c] = #ROL_32(k[c], r);
    return k;
}

inline fn __quarter_round_ref(reg u32[16] k, inline int a b c d) -> reg u32[16]
{
    k = __line_ref(k, a, b, d, 16);
    k = __line_ref(k, c, d, b, 12);
    k = __line_ref(k, a, b, d, 8);
    k = __line_ref(k, c, d, b, 7);
    return k;
}

vsadd %ymm4, %ymm0, %ymm0
vpxor %ymm0, %ymm12, %ymm12
vpshufb (%rsps), %ymm12, %ymm12
vpaddd %ymm12, %ymm8, %ymm8
vpaddd %ymm6, %ymm2, %ymm2
vpxor %ymm8, %ymm4, %ymm4
vpxor %ymm2, %ymm14, %ymm14
vsllld $12, %ymm4, %ymm15
vsrlld $20, %ymm4, %ymm4
vpxor %ymm15, %ymm4, %ymm4
vpshufb (%rsps), %ymm14, %ymm14
vpaddd %ymm4, %ymm0, %ymm0
vpaddd %ymm14, %ymm10, %ymm10
vpxor %ymm0, %ymm12, %ymm12
vpxor %ymm10, %ymm6, %ymm6
vpshufb 32(%rsps), %ymm12, %ymm12
vsllld $12, %ymm6, %ymm15
vsrlld $20, %ymm6, %ymm6
...
**Verified Cryptography Workflow**

- **F** or Coq or EasyCrypt...
- **Potential Implementation Bug**
  - Memory Safety Violation
  - Functional Correctness Flaw
  - Side Channel Vulnerability

- **Deploy Code**
  - Fix and re-verify

**Workflow Diagram**

- **FORMAL SPEC**
- **STANDARD**
- **IMPLEMENTATION**
- **VERIFY**
Good news: For any modern crypto algorithm, there is probably a verified implementation

- You don’t have to sacrifice performance
- Mechanized proofs that you can run and re-run yourself
- You (mostly) don’t have to read or understand the proofs
But... not always easy to use, extend, or combine code from different libraries

- You do need to carefully audit the formal specs, written in tool-specific spec languages like F*, Coq, EasyCrypt
- You do need to safely use their low-level APIs, which often embed subtle pre-conditions
**hacspec: a tool-independent spec language**

**Design Goals**

- **Easy to use** for crypto developers
- **Familiar** language and tools
- **Succinct** specs, like pseudocode
- **Strongly typed** to avoid spec errors
- **Executable** for spec debugging
- **Testable** against RFC test vectors
- **Translations** to formal languages like 
  
  F*, Coq, EasyCrypt, ...
## hacspec: a tool-independent spec language

### Design Goals
- **Easy to use** for crypto developers
- **Familiar** language and tools
- **Succinct** specs, like pseudocode
- **Strongly typed** to avoid spec errors
- **Executable** for spec debugging
- **Testable** against RFC test vectors
- **Translations** to formal languages like F*, Coq, EasyCrypt, ...

### A purely functional subset of Rust
- Safe Rust without external side-effects
- No mutable borrows
- All values are copyable
- Rust tools & development environment
- A library of common abstractions
  - Arbitrary-precision Integers
  - Secret-independent Machine Ints
  - Vectors, Matrices, Polynomials,

---

Language and Toolchain Details: [hacspec.org](http://hacspec.org)
hacspec: purely functional crypto code in Rust

```rust
fn inner_block(st: State) -> State {
    let mut state = st;
    state = chacha20_quarter_round(0, 4, 8, 12, state);
    state = chacha20_quarter_round(1, 5, 9, 13, state);
    state = chacha20_quarter_round(2, 6, 10, 14, state);
    state = chacha20_quarter_round(3, 7, 11, 15, state);
    state = chacha20_quarter_round(0, 5, 10, 15, state);
    state = chacha20_quarter_round(1, 6, 11, 12, state);
    state = chacha20_quarter_round(2, 7, 8, 13, state);
    state = chacha20_quarter_round(3, 4, 9, 14, state);
}
```

ChaCha20 RFC

Call-by-value

State-passing style

ChaCha20 in hacspec
hacspecc: abstract integers for field arithmetic

\[ n = \text{le}_\text{bytes}_\text{to}_\text{num}(\text{msg}[(i-1)*16..(i*16)] | [0x01]) \]
\[ a += n \]
\[ a = (r * a) \mod p \]

Modular 130-bit Prime Field Arithmetic

```rust
pub fn poly1305_encode_block(b: &PolyBlock) -> FieldElement {
    let n = U128_from_le_bytes(U128Word::from_seq(b));
    let f = FieldElement::from_secret_literal(n);
    f + FieldElement::pow2(128)
}

pub fn poly1305_update_block(b: &PolyBlock, (acc,r,s): PolyState) -> PolyState {
    ((poly1305_encode_block(b) + acc) * r, r, s)
}
```

Poly1305 in hacspecc

Poly1305 RFC (update_block)

Modular Arithmetic over User-Defined Field
**hacspecl: secret integers for “constant-time” code**

**Separate Secret and Public Values**
- New types: U8, U32, U64, U128
- Can do arithmetic: +, *, -
- Can do bitwise ops: ^, |, &
- Cannot do division: /, %
- Cannot do comparison: ==, !=, <, ...
- Cannot use as array indexes: x[u]

**Enforces secret independence**
- A “constant-time” discipline
- Important for some crypto specs

```rust
fn chacha20_line(a: StateIdx, b: StateIdx, d: StateIdx, s: usize, mut state: State) -> State {
    state[a] = state[a] + state[b];
    state[d] = state[d] ^ state[a];
    state[d] = state[d].rotate_left(s);
    state
}
```

**AES in hacspecl**

```rust
fn sub_bytes(state: Block) -> Block {
    let mut st = state;
    for i in 0..BLOCKSIZE {
        st[i] = SBOX[U8::declassify(state[i])];
    }
    st
}
```
**hacspec: translation to formal languages**

ChaCha20 in hacspec

```plaintext
pub fn chacha20_quarter_round(a: StateIdx,
   b: StateIdx,
   c: StateIdx,
   d: StateIdx,
   mut state: State,
) -> State {
   state = chacha20_line(a, b, d, 16, state);
   state = chacha20_line(c, d, b, 12, state);
   state = chacha20_line(a, b, d, 8, state);
   chacha20_line(c, d, b, 7, state)
}
```

**F* Spec**

```plaintext
let chacha20_quarter_round (a b c d: state_idx_t) (state: state_t): state_t =
let state:state_t = chacha20_line a b d 16 state in
let state:state_t = chacha20_line c d b 12 state in
let state:state_t = chacha20_line a b d 8 state in
chacha20_line c d b 7 state
```

**Coq Spec**

```plaintext
Definition chacha20_quarter_round (a : int32) (b : int32) (c : int32) (d : int32) (state : State): State :=
let state := chacha20_line a b d 16 state : State in
let state := chacha20_line c d b 12 state : State in
let state := chacha20_line a b d 8 state : State in
chacha20_line c d b 7 state.
```

**EasyCrypt Spec**

```plaintext
proc chacha20_quarter_round(a : int, b : int, c : int, d : int, state : State) = {
    var _res;
    state @$\congchacha20\text{ line } (a, b, d, 16, state);$
    state @$\congchacha20\text{ line } (c, d, b, 12, state);$
    state @$\congchacha20\text{ line } (a, b, d, 8, state);$
    _res @$\congchacha20\text{ line } (c, d, b, 7, state);$
    return _res;
}
```

Active development: [github.com/hacspec](https://github.com/hacspec)
hacspec: towards high-assurance crypto software
hacspect: towards high-assurance crypto software
hacs: towards high-assurance crypto software
libcrux: a library of verified cryptography
libcrux: architecture

LibCrux

Safe APIs  new hacspec constructions  hacspec specs  Reproducible Proofs

HACL Rust Wrapper

Vale Rust Wrapper

libjade Rust Wrapper

HACL* (C)

Vale (x64 asm)

libjade (x64 asm)

AU Curves Fiat (Rust)
Unsafe APIs: Array Constraints

```c
void Hacl_Chacha20Poly1305_32_aead_encrypt(
    uint8_t *k,
    uint8_t *n,
    uint32_t aadlen,
    uint8_t *aad,
    uint32_t mlen,
    uint8_t *m,
    uint8_t *cipher,
    uint8_t *mac
);
```
Verified F* API: Preconditions

let aead_encrypt_st (w:field_spec) =
  key:lbuffer uint8 32ul
  nonce:lbuffer uint8 12ul
  aen:size_t
  aad:lbuffer uint8 aen
  len:size_t
  input:lbuffer uint8 len
  output:lbuffer uint8 len
  tag:lbuffer uint8 16ul

Stack unit
  (requires fun h ->
    live h key \ live h nonce \ live h aad \
    live h input \ live h output \ live h tag \
    disjoint key output \ disjoint nonce output \ disjoint key tag \ disjoint nonce tag \ disjoint output tag \ eq_or_disjoint input output \ disjoint aad output)
Verified F* API: Preconditions

let aead_encrypt_st (w:field_spec) =
  key:buffer uint8 32ul
-> nonce:buffer uint8 12ul
-> alen:size_t
-> aad:buffer uint8 alen
-> len:size_t
-> input:buffer uint8 len
-> output:buffer uint8 len
-> tag:buffer uint8 16ul ->
Stack unit
(requires fun h ->
  live h key ∧ live h nonce ∧ live h aad ∧
live h input ∧ live h output ∧ live h tag ∧
  disjoint key output ∧ disjoint nonce output ∧
  disjoint key tag ∧ disjoint nonce tag ∧
  disjoint output tag ∧ eq_or_disjoint input output ∧
  disjoint aad output)
libcrux: Typed Rust APIs

type Chacha20Key = [u8; 32];
type Nonce = [u8; 12];
type Tag = [u8; 16];

fn encrypt(
    key: &Chacha20Key,
    msg_ctxt: &mut [u8],
    nonce: Nonce,
    aad: &[u8]
) -> Tag
libcrux: supported algorithms & perf
<table>
<thead>
<tr>
<th>Crypto Standard</th>
<th>Platforms</th>
<th>Specs</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ECDH</strong></td>
<td>x25519</td>
<td>hacspec, F*</td>
<td>HACL*, Vale</td>
</tr>
<tr>
<td></td>
<td>P256</td>
<td>hacspec, F*</td>
<td>HACL*</td>
</tr>
<tr>
<td><strong>AEAD</strong></td>
<td>Chacha20Poly1305</td>
<td>hacspec, F*, EasyCrypt</td>
<td>HACL*, libjade</td>
</tr>
<tr>
<td></td>
<td>AES-GCM</td>
<td>hacspec, F*</td>
<td>Vale</td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td>Ed25519</td>
<td>hacspec, F*</td>
<td>HACL*</td>
</tr>
<tr>
<td></td>
<td>ECDSA P256</td>
<td>hacspec, F*</td>
<td>HACL*</td>
</tr>
<tr>
<td></td>
<td>BLS12-381</td>
<td>hacspec, Coq</td>
<td>AUCurves</td>
</tr>
<tr>
<td><strong>Hash</strong></td>
<td>Blake2</td>
<td>hacspec, F*</td>
<td>HACL*</td>
</tr>
<tr>
<td></td>
<td>SHA2</td>
<td>hacspec, F*</td>
<td>HACL*</td>
</tr>
<tr>
<td></td>
<td>SHA3</td>
<td>hacspec, F*, EasyCrypt</td>
<td>HACL*, libjade</td>
</tr>
<tr>
<td><strong>HKDF, HMAC</strong></td>
<td></td>
<td>hacspec, F*</td>
<td>HACL*</td>
</tr>
<tr>
<td><strong>HPKE</strong></td>
<td></td>
<td>hacspec</td>
<td>Hacspect</td>
</tr>
</tbody>
</table>
## libcrux: performance

<table>
<thead>
<tr>
<th></th>
<th>libcrux</th>
<th>Rust Crypto</th>
<th>Ring</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sha3 256</td>
<td>574.39 MiB/s</td>
<td>573.89 MiB/s</td>
<td>unsupported</td>
<td>625.37 MiB/s</td>
</tr>
<tr>
<td>x25519</td>
<td>30.320 µs</td>
<td>35.465 µs</td>
<td>30.363 µs</td>
<td>32.272 µs</td>
</tr>
</tbody>
</table>

**Intel Kaby Lake (ADX, AVX2)**

<table>
<thead>
<tr>
<th></th>
<th>libcrux</th>
<th>Rust Crypto</th>
<th>Ring</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sha3 256</td>
<td>337.67 MiB/s</td>
<td>275.05 MiB/s</td>
<td>unsupported</td>
<td>322.21 MiB/s</td>
</tr>
<tr>
<td>x25519</td>
<td>37.640 µs</td>
<td>67.660 µs</td>
<td>71.236 µs</td>
<td>48.620 µs</td>
</tr>
</tbody>
</table>

**Apple Arm M1 Pro (Neon)**
RFC 9180
Hybrid Public Key Encryption

https://tech.cryspen.com/hpke-spec/hpke/index.html
HPKE: Construction

- HPKE
  - KEM
    - DHKEM
      - P-256, HKDF-SHA256
    - ...
  - KDF
    - HKDF
      - HKDF SHA256
    - ...
  - AEAD
    - AES-GCM 128
    - ...

## HPKE code performance: hacspec vs. stateful Rust

<table>
<thead>
<tr>
<th></th>
<th>hacspec HPKE</th>
<th>Rust HPKE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup Sender</td>
<td>79.9 μs</td>
<td>68 μs</td>
</tr>
<tr>
<td>Setup Receiver</td>
<td>76 μs</td>
<td>54.4 μs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>libcrux</th>
<th>RustCrypto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sha2 256</td>
<td>311.76 MiB/s</td>
<td>319.10 MiB/s</td>
</tr>
<tr>
<td>x25519</td>
<td>30.320 μs</td>
<td>35.465 μs</td>
</tr>
<tr>
<td>x25519 base</td>
<td>30.218 μs</td>
<td>11.812 μs</td>
</tr>
<tr>
<td>ChaCha20Poly1305</td>
<td>758.89 MiB/s</td>
<td>249.33 MiB/s</td>
</tr>
</tbody>
</table>
Ongoing and Future Work
The Last Yard: linking hacspec to security proofs

https://eprint.iacr.org/2023/185
## Verification Tools: more proof backends for hacspec

<table>
<thead>
<tr>
<th>Security Analysis Tools</th>
<th>Program Verification Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSProve: modular crypto proofs</td>
<td>QuickCheck: logical spec testing</td>
</tr>
<tr>
<td>EasyCrypt: verified constructions</td>
<td>Creusot: verifying spec contracts</td>
</tr>
<tr>
<td>ProVerif: symbolic protocol proofs</td>
<td>Aeneas: verifying Rust code</td>
</tr>
<tr>
<td>CryptoVerif: verified protocols</td>
<td>LEAN: verification framework</td>
</tr>
<tr>
<td>Squirrel: protocol verifier</td>
<td>&lt;Your favourite prover here&gt;</td>
</tr>
</tbody>
</table>
Conclusions

- **Fast verified code** is available today for most modern crypto algorithms
  - + some post-quantum crypto; **Future**: verified code for ZKP, FHE, MPC, ...
  - Most code in C or Intel assembly; **Ongoing**: Rust, ARM assembly, ...

- **hacspec** can be used as a common spec language for multiple tools/libraries
  - **Ongoing**: adding new Rust features, new proof backends, linking with Rust verifiers, ...
  - **Try it yourself**: hacspec.org

- **libcrux** provides safe Rust APIs to multiple verified crypto libraries
  - **Ongoing**: recipes for integrating new verified crypto from various research projects
  - **Try it yourself**: libcrux.org
Thanks!

- HACL*: https://github.com/hacl-star/hacl-star
- Vale: https://github.com/ValeLang/Vale
- libjade: https://github.com/formosa-crypto/libjade
- AUCurves: https://github.com/AU-COBRA/AUCurves
- hacspec: https://github.com/hacspec/hacspec
- libcrux: https://github.com/cryspen/libcrux